Short Introduction to Python

Machine Learning – Laboratory

Battista Biggio

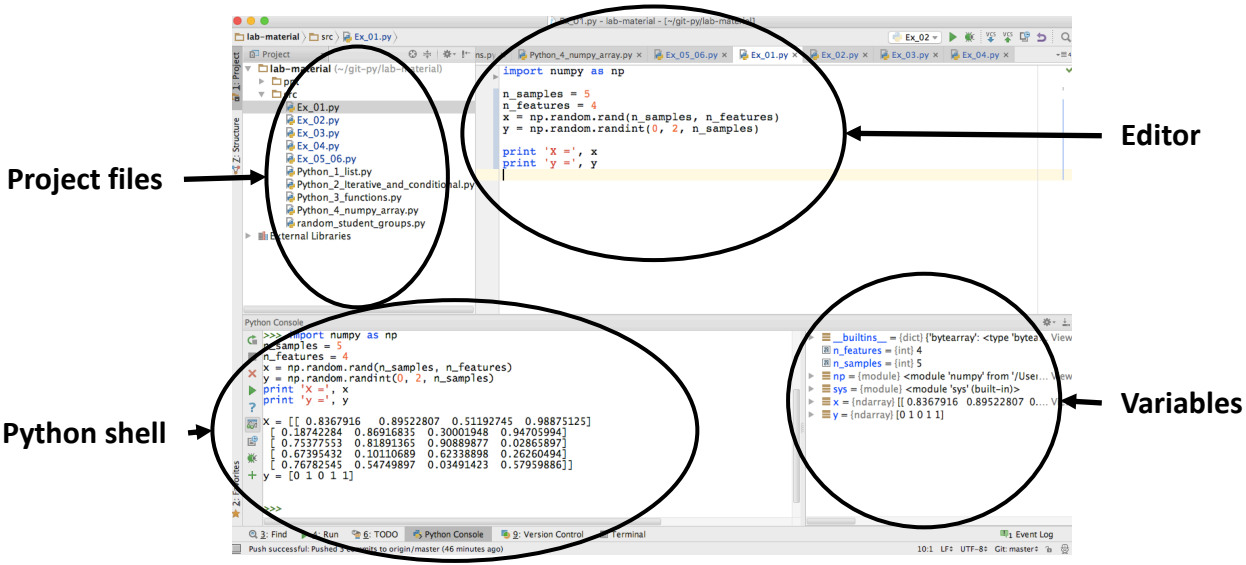
Department of Electrical and Electronic Engineering
University of Cagliari, Italy

Why Python?

- Reasonable trade-off between high-level language abstraction and efficiency
  - C/C++ is more efficient, but requires lower abstraction (e.g., memory allocation)
  - Matlab is maybe easier to program, but typically less efficient
- Python is a high-level programming language providing several third-party libraries that normally wrap C/C++ (efficient) code
- Availability of machine-learning libraries (scikit-learn.org, pytorch.org)

PyCharm

- Development IDE https://www.jetbrains.com/pycharm/
- Easy to setup and use
  - Integration with repositories / versioning mechanisms (subversion, git)
- Compliant to PEP8 programming guidelines (after configuration)



Laboratory Outline (Tentative)

- Python basics (2 hours, today)
- Dataset manipulation
- Implementation of learning and classification algorithms
  - Nearest Mean Classifier (NMC)
- Performance evaluation using cross validation
- Parameter estimation using cross validation
- Evaluation of learning algorithms
  - Support Vector Machines (SVMs), decision trees, neural networks
- Application examples
  - Design of a complete machine-learning system
  - Student competitions

Student Evaluation: Laboratory assignment (4 hours)

We will use Python 3 during the course

Python Basics

Data Types

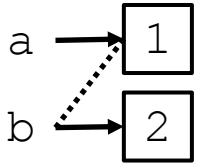
- Base elements
  - **int** (e.g., 1), **float** (e.g., 1.0), **strings** (e.g., "a word" or 'a word')
- Lists
  - flexible arrays, can be modified *in place (mutable objects)*
  - `a = [ 'wolf', 'cat', 'horse', 1, 1.0, ['a', 'b'], 3 ]`
- Tuples
  - similar to lists, but *immutable* (faster indexing with hashing)
  - `a = ('wolf', 'cat')`
- Dictionaries (hash tables)
  - key-value pairs (keys are *immutable*, values are *mutable*)
  - `a = {'key1': 0, 'key2': 1}`
- Numerical arrays (we will see more on numpy arrays later on)

Basic Operations

- Important: everything in Python is a variable
  - including functions, classes, modules

- Immutable objects can not be modified (reference changes!)

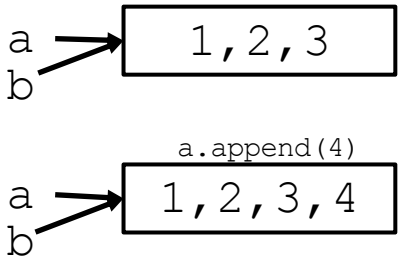
```
>>> a = 1
>>> b = a
>>> b = 2
>>> print a, b
1 2
```



- Mutable objects (like ***lists***) can be modified (by ***in-place*** functions)

```
>>> a = [1, 2, 3]
>>> b = a

>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```



Basic Operations (Lists)

- Creation

```
>>> a = [1, 2, 3]
>>> a = range(0,5)
>>> print a
[0, 1, 2, 3, 4]
>>> a = range(0,5,2)
>>> print a
[0, 2, 4]
```

- Indexing

```
>>> a = [1, 2, 3]
>>> print a[0], a[1], a[-1]
1 2 3
>>> print a[1:]  # slicing from index 1 to end
[2,3]
>>> print a[:2]  # slicing from index 0 to 2 (excluded)
[1,2]
```

Basic Operations (Lists)

- Reference and copy

```
>>> a = [1, 2, 3]
>>> b = a  # reference assignment. a and b are modified
>>> a[0] = 2
>>> print a, b
[2, 2, 3] [2, 2, 3]

>>> b = a[:]  # this is a copy. now a and b are different
>>> a[0] = 2
>>> print a, b
[2, 2, 3] [1, 2, 3]
```

- Other operations

```
>>> a = [3, 1, 2]
>>> len(a)  # returns number of elements in list
3
>>> a.sort()  # sorts list in ascending order [1, 2, 3]
>>> a.append(4)  # appends element at end -> [1, 2, 3, 4]
```

Control Structures

- **if** *condition*:
  statements

  elif *condition*:
  statements

  else:
  statements

```
a=0
if a<0:
    print 'a is negative'
elif a>0:
    print 'a is positive'
else:
    print 'a is zero'
```

- **for** *var* **in** *sequence:*
  statements

```
a=['a','b','c']
for elem in a:
    print elem
```

- **while** *condition is True:*
  statements

```
a = 5
while a > 0:
    a = a - 1
    print a
```

Functions / Procedures

```
def name(arg1, arg2, ...):
    """documentation"""  # optional doc string
    statements
    return  # from procedure
    return expression  # from function
```

```
def gcd(a, b):
    """greatest common divisor"""
    while a != 0:
        a, b = b % a, a  # parallel assignment
    return b
```

```
>>> print gcd(12, 20)
4
```

Numpy Arrays (array)

```
import numpy as np

a = np.array([[1, 2, 3],
              [4, 5, 6]])
>>> a.shape
(2, 3)

>>> a.dtype
dtype('int64')

n_rows, n_cols = 2, 4
a = np.zeros(shape=(n_rows, n_cols))
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Other Array Creation Methods

```
a = np.ones(shape=(n_rows, n_cols))  # creates matrix of ones

a = np.eye(n_rows, n_cols)  # creates identity matrix
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.]])

# random numbers from Normal distribution
# with zero mean and unit variance
a = np.random.randn(n_rows, n_cols)

# random numbers from Uniform distribution in [0,1]
a = np.random.rand(n_rows, n_cols)
>>> a
[[ 0.02413213  0.27179266  0.8904404   0.78110291]
 [ 0.66329305  0.57972581  0.19168138  0.44013989]]

>>> np.random.randint(0, 5, [n_rows, n_cols])  # random integers
[[ 0  2  1  0]
 [ 1  4  3  1]]
```

Array Indexing

- This can be rather complicated
  - https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html
- Let's keep it simple, using some simple rules
  - Index arrays with shape = (n,) (flat arrays, vectors) with a single index
  - Index matrices with shape = (n, m) using two indices
  - In general, if shape.size == K, we should use K indices

```
a = np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> a[0, 0]  # picks the first element (returns float)
1.0
>>> a[0, :]  # picks the first row (returns flat array)
array([ 1.,  0.,  0.])
>>> a[:, 1]  # picks the second column (returns flat array)
array([ 0.,  1.,  0.])
```

Array Indexing

```
a = np.eye(3)
>>> a
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

>>> a[0:2, 0:2]  #selects submatrix with slicing operators
array([[ 1.,  0.],
       [ 0.,  1.]])

b = np.array([1, 1, 0])
>>> a[b==1, :] # b used to index rows (picks first and second here)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

Other Operation on Arrays

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

>>> np.vstack((a, b))  # stack rows (vertical stacking)
array([[1, 2, 3],
       [4, 5, 6]])

>>> np.hstack((a, b)) # stack columns (horizontal stacking)
array([1, 2, 3, 4, 5, 6])

>>> a.dot(b)  # scalar product
32

# other operations include: reshape, transpose, min/max, etc.
# we will see more througout the course, when required
```

Exercises

Exercise 1

- Create an ndarray `x` of `shape=(5,4)` with random numbers
  - Each of the 5 rows represents a sample with 4 features
- Create a flat ndarray `y` of `shape=(5, )`, whose elements are 0 and 1
  - Each element is the class label of the corresponding sample in x
- This is a simple example of a two-class dataset!

Hint: use `np.random.rand(...)` and `np.random.randint(...)`

```
x = [[ 0.33990211  0.94182274  0.66611658  0.72773846]
     [ 0.33281557  0.24280422  0.3627702   0.40495032]
     [ 0.5016927   0.29465024  0.61690932  0.25302243]
     [ 0.01744464  0.82521145  0.82226041  0.89858553]
     [ 0.33772606  0.17433791  0.7705529   0.11211808]]
y = [0 0 1 1 1]
```

Exercise 1: Solution

```
import numpy as np

n_samples = 5
n_features = 4

x = np.random.rand(n_samples, n_features)
y = np.random.randint(0, 2, n_samples)

print("x =", x)
print("y =", y)
```

Exercise 2

- Define a function `extract_subset(x, y, y0)` that takes as input:
  - the feature matrix `x`, and the labels `y` from the previous exercise
  - a target class `y0` (i.e., either `y0=0` or `y0=1`)
- and returns a feature matrix containing only samples belonging to `y0`

Hint: use *array indexing* with `y==y0`

```
x = [[ 0.33990211  0.94182274  0.66611658  0.72773846]
     [ 0.33281557  0.24280422  0.3627702   0.80495032]
     [ 0.5016927   0.29465024  0.61690932  0.25302243]
     [ 0.01744464  0.82521145  0.82226041  0.89858553]
     [ 0.33772606  0.17433791  0.7705529   0.11211808]]
y = [0 0 1 1 1]

>>> extract_subset(x, y, 0)
array([[ 0.33990211  0.94182274  0.66611658  0.72773846],
       [ 0.33281557  0.24280422  0.3627702   0.80495032]])
```

Exercise 2: Solution (cont'd from Exercise 1)

```
[code from Exercise 1]

def extract_subset(x, y, y0):
    # we should do some checks to validate inputs
    # e.g., to check if y0 is 0 or 1, etc.
    # but we will address this problem later on
    return x[y == y0, :]


x0 = extract_subset(x, y, 0)
print('x0 =', x0)
```

Exercise 3

- Define a function `min_feature_values(x)` that returns the minimum value of each feature in `x`
- Apply it on the previously-extracted samples `x0` of class 0

Hint: use `np.min()` with a proper `axis` value

```
x0 = [[ 0.33990211  0.94182274  0.66611658  0.72773846]
      [ 0.33281557  0.24280422  0.3627702   0.80495032]]

>>> min_feature_values(x0)
array([0.33281557  0.24280422  0.3627702 0.72773846])
```

Exercise 3: Solution

```
[code from Exercise 2]

def min_feature_values(x):
    return np.min(x, axis=0)

print('min. feat. values: ', min_feature_values(x0))
```

Exercise 4

- Define a function **`make_gaussian_dataset`** `(n0, n1, mu0, mu1)` that generates a two-class Gaussian dataset in a bi-dimensional space
  - `n0, n1` are the number of samples for class 0 and class 1
  - `mu0, mu1` are the means of the two Gaussians (one per class)

- We consider only Gaussian distributions with covariance equal to the identity matrix here for simplicity

- The function returns the corresponding feature matrix and labels `x, y`

Hints:

- use `np.random.randn(...)` to generate random numbers from a standard Gaussian distribution, with zero mean and unit variance, and then transform them to have a different mean (repeat twice, one per class)
- use `np.ones(...)` and/or `np.zeros(...)` for class labels
- use `np.vstack(...)` and `np.hstack(...)` to concatenate arrays

Exercise 4: Solution

```
import numpy as np

def make_gaussian_dataset(n0, n1, mu0, mu1):
    """ Creates a 2-class 2-dimensional Gaussian dataset. """
   d = 2  # hard-coded for convenience, we will improve this later on

   x0 = np.random.randn(n0, d) + mu0  # uses broadcasting...
   x1 = np.random.randn(n1, d) + mu1

   # sample labels
   y0 = np.zeros(n0)
   y1 = np.ones(n1)

   # concatenate data and labels
   x = np.vstack((x0, x1))
   y = np.hstack((y0, y1))

   return x, y


# generate data with 10 samples/class, and means [-1,-1], [1, 1]
xn, yn = make_gaussian_dataset(10, 10, [-1, -1], [+1, +1])
print('xn: ', xn)
print('yn: ', yn)
```